第二十八节 分布式消息与事件

Spring Cloud Stream

通过 Spring Cloud Stream 屏蔽消息（事件）中间件 API 的细节，提供统一 API 接口进行消息（事件）发布和订阅。

3.0 之前的实现

文档：https://docs.spring.io/spring-cloud-stream/docs/Fishtown.M3/reference/htmlsingle/#spring-cloud-stream-overview-introducing

基于 Spring Message 接口实现（底层）

- org.springframework.messaging.MessageChannel
- org.springframework.messaging.SubscribableChannel
- org.springframework.messaging.Message

基于 Spring Integration 接口实现（中层）

- @org.springframework.integration.annotation.ServiceActivator

基于 Spring Cloud Stream 接口实现（高层）

- @org.springframework.cloud.stream.annotation.StreamListener

接口实现

@Source

@Sink

@Processor

消费端注解

- @Input

提供方注解

- @Output

3.0 + 的实现

增加 Java @FunctionalInterface 实现

- @java.util.funciton.Consumer
- @java.util.funciton.Supplier
- @java.util.function.Function

名词含义

消息

HTTP 消息

RPC 消息

MOM 消息

类似：事件

底层实现

Spring Integration

- HTTP 消息(事件)
- FTP
- File
- Message
- RPC

技术衍生

Spring Cloud Data Flow

相当于 Spring Cloud Stream 应用作为流式计算的节点，

数据提供（来源）：Source

数据消费（处理）：Sink

数据中间处理：Processor

站在 Spring Cloud Stream 角度，Source 可能是一个数据通道

站在 Spring Cloud Data Flow 角度，Source 是一个 Spring Cloud Stream 应用。

一个 Spring Cloud Data Flow 集群，对应多个 Spring Cloud Stream 应用，

一个 Spring Cloud Stream 应用，对应多一个 Source、Sink、Processor（可选）组件

Spring Cloud Bus

使用场景

Spring Cloud Bus 事件发送方将 Spring 远程事件封装为 Spring 消息，并利用 Spring Cloud Stream 传输消息到消息中间件。对于事件消费方则利用 Spring Cloud Stream 订阅事件消息，并且转化为 Spring 事件，利用 Spring 事件/监听机制来处理事件。

核心目标，Spring Cloud Bus Source 和 Sink 均使用 Spring 事件来完成操作，无需关心事件在分布式环境中存储细节。

底层实现

- Spring Cloud Stream
- Spring Event

具体实现

引入不同的 Spring Cloud Bus 实现，实际是引入 Spring Cloud Stream Binder 实现

核心逻辑

- Spring Cloud Bus

事件发布者（Source）将 Spring Event 转化为 Spring Message，通过 Spring Cloud Stream Binder 通讯，发送到消息中间件，在通过事件监听者（Sink）订阅相关事件的 Topic，再讲 Spring Message 转化为 Spring Event。

事件触发需要通过 Spring Cloud Bus Endpoint 激活或者相关 API 调用。

为什么要转化 Spring 事件?

传统的 Spring 事件/监听作用在单个 Spring 应用，Spring Cloud Bus 将 Spring 事件分布式化。

事件目标

目标应用、服务（集群）

目标应用实例、服务实例（节点）

Spring Cloud Bus 远程事件 - org.springframework.cloud.bus.event.RemoteApplicationEvent

基于 Spring 事件类 - org.springframework.context.ApplicationEvent 实现的抽象基类，它被 Jackson 注解描述：

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME,
property = "type")
@JsonIgnoreProperties("source")
public abstract class RemoteApplicationEvent
extends ApplicationEvent {
}
```

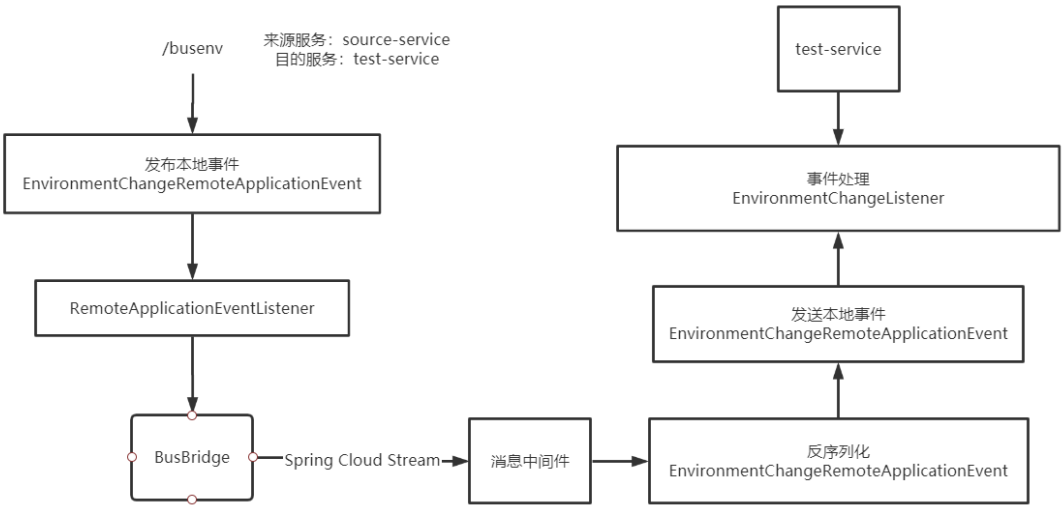
来源服务 - originService

目标服务 - destinationService

子类

- org.springframework.cloud.bus.event.EnvironmentChangeRemoteApplicationEvent - 配置变化事件
- org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent - @RefreshScope 事件
- org.springframework.cloud.bus.event.UnknownRemoteApplicationEvent - 未知事件

处理流程



Java Function

核心接口

函数标注注解 - @java.lang.FunctionalInterface

被标记的接口中有且仅有一个抽象方法，不包含 Object 继承的 public 方法，并且也排除 default 方法。

提供者接口 - java.util.function.Supplier / java.util.function.BiSupplier

消费者接口 - java.util.function.Consumer / java.util.function.BiConsumer

函数接口 - java.util.function.Function / java.util.function.BiFunction

处理接口 - java.lang.Runnable

判断接口 - java.util.function.Predicate

接口设计

特色

链式设计

比如 Consumer：

```
@FunctionalInterface
public interface Consumer<T> {
    ...
    /**
     * Returns a composed {@code Consumer} that
performs, in sequence, this
     * operation followed by the {@code after}
operation. If performing either
```

```
     * operation throws an exception, it is
relayed to the caller of the
     * composed operation.  If performing this
operation throws an exception,
     * the {@code after} operation will not be
performed.
     *
     * @param after the operation to perform
after this operation
     * @return a composed {@code Consumer} that
performs in sequence this
     * operation followed by the {@code after}
operation
     * @throws NullPointerException if {@code
after} is null
     */
    default Consumer<T> andThen(Consumer<?
super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t);
after.accept(t); };
    }
}
```

```
public interface Function<T, R> {
    ...
    default <V> Function<V, R>
compose(Function<? super V, ? extends T>
before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }
    ...
    default <V> Function<T, V>
andThen(Function<? super R, ? extends V> after)
{
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }
}
```

不足

- Java 内建的函数接口均无法抛出 checked 异常
- JDK 内部运用不是特别广泛（参考 Spring Framework 做得比较彻底）

Spring Cloud Function

相关议题

流行架构

云原生

云原生架构并不太关注于功能实现，关于软件生命周期是否在云平台来完成：

- Codebase
- Coding
- CI/CD
  - 金丝雀发布/灰度发布/蓝绿发布
- 项目管理
  - 资源管理
  - 团队管理
  - 资产管理
- 扩展架构
  - 微服务架构
  - Severless 架构
    - FaaS - Function
    - 可选地支持 Reactive
      - 编程模型：Functional
      - 并发模型

        - 同步和异步 API 统一
        - 非阻塞

JVM 平台

- 优势
    - 一次编译，到处运行
    - 准实时性能
        - 商业版： RealTime JVM（https://www.oracle.com/technical-resources/articles/javase/jsr-1.html）
    - 自动内管管理
    - 高度抽象和统一 API
        - 文件系统
        - 网络系统
        - 集合框架
    - JIT（Just-In Time）编译：减少编译时时间
- 不足
    - 内存开销（统一数据结构）
    - GC overhead（STW）
    - JIT（Just-In Time）编译：增加了运行时时间
- 痛点：运行时可能存在停顿，启动时间需要预热（Class Loading，方法编译）
    - 解决方案：GraalVM（Oracle）
    - 其他语言实现 JVM 语法

Stream 平台

- Spring Cloud Stream
- Kafka Stream
- CentOS Stream
- Flink
    - Spark
    - Storm

Aone 平台

Spring 技术栈发展

早期 Java EE 整合框架

中期 Java 现代技术栈

- Java SE
- Java EE
- Java 微服务

现代 Java 未来技术栈

- Function
- Reactive
- Stream
- Native

终极目的：弱化语言差异，强调编程统一

利用已有服务，让服务原子化，让服务可编排

Spring 技术栈抽象内容

Spring Framework

- Spring AOP 抽象
- Spring @Inject 或 @Resource 依赖注入
- Spring Transaction 抽象（事务）
- Spring Environment 抽象（配置）
- Spring Caching 抽象（缓存）
- Spring Web 客户端抽象（Web）
  - RestTemplate
  - WebClient

Spring Boot

- Actuator 抽象
  - Endpoint 抽象
  - Metris 抽象
  - Health 抽象

Spring Cloud

- 服务注册与发现抽象
  - DiscoveryClient
  - ServiceRegistry
- 负载均衡抽象
- 服务熔断抽象
- 分布式消息抽象
- 函数抽象
- 分布式任务抽象
- 分布式事件抽象
- 分布式跟踪抽象
- 分布式锁（暂时缺少）
- 分布式事务（暂时缺少）

作业

利用 Redis 实现 Spring Cloud Bus 中的 BusBridge，避免强依赖于 Spring Cloud Stream

难点：处理消息订阅

目的：

- 回顾 Spring 事件
- 理解 Spring Cloud Bus 架构
- 理解 Spring Cloud Stream